A Near-Optimal Algorithm for a Locality-Maximizing Placement Problem

Fan Chung, Ronald Graham, Ranjita Bhagwan, Stefan Savage, and Geoffrey M. Voelker

Department of Computer Science and Engineering
University of California, San Diego
{fan, rgraham, rbhagwan, savage, voelker}@cs.ucsd.edu

Abstract. The effectiveness of a distributed system hinges on the manner in which tasks and data are assigned to the underlying system resources. Moreover, today's large-scale distributed systems must accommodate heterogeneity in both the offered load and in the makeup of the available storage and compute capacity. The ideal resource assignment must balance the utilization of the underlying system against the loss of locality incurred when individual tasks or data objects are fragmented among several servers. In this paper we describe this *locality-maximizing placement* problem and show that an optimal solution is NP-hard. We then describe a polynomial-time algorithm that generates a placement within an additive constant of two from optimal.

1 Introduction

In recent years, the field of global-scale distributed systems has seen tremendous growth. For example, peer-to-peer storage systems such as OceanStore [KBC+00], CFS [DKK+01], PAST [RD01] and IVY [MMGC02] provide persistent data access using globally distributed and highly heterogeneous storage resources. Similarly, distributed computing efforts such as the Computational Grid [FK99], SETI@home [Set], Entropia [Ent] and BOINC [Boi] envision the use of a widely distributed computing platform for a variety of resource-intensive applications.

In these systems, tasks and data objects are often too large to be assigned to a single node and the system must fragment them among several servers. As a result, the system must balance the utilization of the underlying system resources against the loss of locality incurred when individual tasks or data objects are fragmented. We call this trade-off of system utilization and resource locality the *locality-maximizing placement* problem.

For example, many peer-to-peer storage systems manage very large objects such as MPEG-encoded movies. However, individual hosts taking part in such systems may not be willing or able to store such large files in their entirety. Therefore, the system must partition a movie into separate fragments, each stored on a separate host. However, to recover the movie in its entirety, the hosts storing all of the fragments need to be available at the same time. As a result, the availability of a movie decreases as the number of fragments used to

store it increases. To maximize availability, the system must minimize the number of times files are fragmented, while still assigning all objects to servers.

Distributed computing applications also encounter a similar problem. For many applications it is important to schedule an application's tasks to meet a given timing constraint. However, one or more individual tasks may require more processing power than any single hosts can provide. As a result, tasks must be split across hosts to meet the response time constraint. However, splitting incurs its own costs, such as communication between the different task fragments, replication of data required for the task to execute across all hosts, and data aggregation once the task completes. Therefore the system must select a schedule that makes the minimum necessary number of splits while still scheduling all tasks successfully.

These problems, instances of a locality-maximizing placement problem, represent a specific kind of bin-packing problem that can be stated as follows. Given are a fixed set of bins of varying sizes as well as items of varying sizes that need to be placed into the bins. The sizes of the items may be too large to fit into individual bins, and so they may need to be split into fragments to fit into the bins. However, fragmenting of items causes a "loss of locality" for that item. In the peer-to-peer storage problem mentioned above, the loss of locality decreases file availability. In the distributed computing problem, it leads to higher communication and storage overheads. In general, the more fragments for an item, the worse its locality.

One potential solution to the problem is to find a packing that maximizes the average (or total) locality of the items, i.e., minimizes the average (or total) number of item fragments. However, minimizing the average does not bound the worst-case number of fragments of individual items, and so some items could have a large loss in locality due to extensive fragmentation.

A more desirable solution of the problem is a packing that maximizes the minimum locality over all items, or in other words, minimizes the maximum number of fragments made of *any* item. This ensures a minimum locality for all the items. In the first example, the system would fragment and store files such that it maximizes the minimum file availability. In the second example, the system would minimize the maximum communication and storage overhead.

In this paper, we show that determining the optimal solution to the locality-maximization placement problem is NP-hard. We then describe a polynomial-time algorithm that generates a placement within an additive constant of two from optimal. We also show experimental results obtained from applying our algorithm to a large number of simulated systems.

The rest of the paper is organized as follows. Section 2 provides the formal problem definition, and shows that solving it is NP-hard. In Section 3, we derive a lower-bound for the optimal solution of the problem. We also state and prove claims that will be used in later

sections in the description and verification of our algorithm. In Section 4, we describe our algorithm, calculate its running-time, and provide experimental results. Finally, Section 5 summarizes the contributions of this paper.

2 Problem definition

Let $I = (I_1, I_2, \ldots, I_m)$ be the set of items and let $B = (B_1, B_2, \ldots, B_n)$ be the set of bins. Also, let $|I_i|$ denote the size of item I_i, and let $|B_j|$ denote the capacity of bin B_j. Without loss of generality, we assume that the cumulative sizes of the bins equals the cumulative size of the items, *i.e.*, $\sum_i |I_i| = \sum_j |B_j|$.

We define a packing P as an assignment of every item to the set of bins, given that each item can be fragmented across multiple bins, and similarly, a bin can hold multiple item fragments. For a given packing P, we define $h_P(I_i) = h(I_i) :=$ number of bins "hit" by I_i, or, in other words, the number of bins that contain a fragment of the item I_i. Similarly, $h_P(I_i \cup I_j) =$ number of bins hit by I_i and I_j, etc.

We want to find a packing that minimizes the maximum number of fragments made of an item, which is equal to the maximum number of bins hit by an item. So, define

$$OPT(I, B) = \min_P \max_{1 \le k \le m} h_P(I_k)$$

where P ranges over all packings of (I, B).

2.1 OPT(I, B) is NP-hard

We point out that determining OPT(I, B) in general is an NP-hard problem. To see this, we consider the following known NP-hard bin packing problem BP ([Kar72], also see page 223 of [GJ75]). We will reduce the problem BP to a special case of our problem, specifically to the question "Is OPT$(I, B) = 1$?"

Problem BP

Input : Set S of n positive integers $s_1, s_2, \ldots, s_n$ with sum $= 2\sigma$.

Question: Is there a subset of S with sum $= \sigma$.

We can use this input data to construct a special case of our problem by defining

$I = \{I_1, I_2, \ldots, I_n\}$ with $|I_k| = s_k, 1 \le k \le n$, and
$B = \{B_1, B_2\}$ with $|B_1| = |B_2| = \sigma$.

Then OPT$(I, B) = 1$ if and only if the answer to the BP question is yes.

3 Basic facts

Since the locality-maximizing placement problem is NP-Hard, we have developed a polynomial-time algorithm that provides a solution that is within an additive constant of 2 from the optimal. In this section, we provide several definitions and prove various facts that are required for an explanation of our algorithm. We first derive a lower bound to the optimal solution to the above problem. We then make additional claims that shall be used in future sections and in the proposed algorithm.

We assume that the items and bins are sorted in non-ascending order, that is (after relabeling),

$$|I_1| \geq |I_2| \geq \ldots \geq |I_m|$$

$$|B_1| \geq |B_2| \geq \ldots \geq |B_n|$$

We next create the "canonical packing" C by assigning bins in the sorted order to the items, also in a sorted order. For example, as shown in Figure 1, B_1, B_2 and B_3 are assigned to I_1. The rest of B_3 is filled by I_2, and so on. So I_1 "hits" B_1, B_2 and B_3, making $h(I_1)$ equal to 3. I_2 hits B_3, B_4 and B_5, and therefore $h(I_2)$ is also equal to 3. Since I_1 and I_2 together hit B_1, B_2, B_3, B_4 and B_5, $h(I_1 \cup I_2)$ is 5. We can always assume that in representing the bins into which I_i is packed, each bin B_j is represented by an interval of I_i. The ordering of these bins, or intervals, within I_i, is irrelevant.

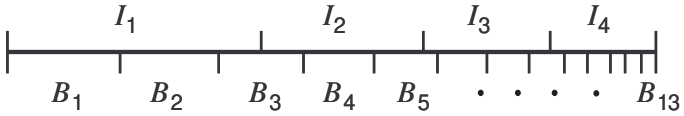

Fig. 1. An example problem with 4 items and 13 bins. This shows the canonical packing C with the items and bins both sorted in non-ascending size order.

For $1 \leq k \leq m$, we define

$$\tau(k) := \lceil \frac{1}{k} h(I_1 \cup \ldots \cup I_k) \rceil$$

$$\text{and} \quad \tau := max_k \tau(k)$$

where $h = h_C$, and C is the canonical packing.

Going back to the example of Figure 1,

$$\begin{aligned} h(I_1) = 3, \quad h(I_2) = 3, \quad h(I_3) = 4, \quad h(I_4) = 6, \\ \tau(1) &= 3, \\ \tau(2) &= \lceil 5/2 \rceil = 3, \\ \tau(3) &= \lceil 8/3 \rceil = 3, \\ \tau(4) &= \lceil 13/4 \rceil = 4, \\ \tau &= 4. \end{aligned}$$

Claim 1 $\mathrm{OPT}(I, B) \geq \tau$.

Proof. We shall show by contradiction that it is not possible for $\mathrm{OPT}(I, B)$ to be less than τ, and consequently τ is a lower bound to the solution of our packing problem. Suppose

$$OPT(I, B) < \tau \tag{1}$$

Let us choose k such that

$$\tau(k) = \lceil \frac{1}{k} h_C(I_1 \cup \ldots \cup I_k) \rceil = \tau$$

Thus, k corresponds to the maximum value of $\tau(i)$, which is equal to τ. Let $h = h_C(I_1 \cup \ldots \cup I_k)$. Now, from the definition of τ, we can say that there exists I_i, $i \leq k$, for which

$$h(I_i) \geq \tau \tag{2}$$

However, for assumption (1) to hold, the number of bins hit by any item must be less than τ. To achieve this, we need to change the packing from the canonical packing C.

If we change the ordering of bins within the first k items, thus obtaining a new packing, inequality (2) would still hold, since $h(I_1 \cup \ldots \cup I_k)$ would remain the same. So for the new packing, we will need to use some bins for the first k items that have not already been hit by them in the canonical packing. But since $B_1, \ldots, B_{h-1}$ are the *largest* $h - 1$ bins, no other set of $h - 1$ bins can hold items $I_1 \cup \ldots \cup I_k$. Hence, by changing the ordering, we can only increase the number of bins hit by the first k items. Thus, there is no way that we can satisfy (1), which is a contradiction. □

Claim 2 For the canonical packing C,

$$h_C(I_1) + \ldots + h_C(I_k) \leq h_C(I_1 \cup \ldots \cup I_k) + k - 1$$

for $1 \leq k \leq m$.

Proof. There are a total of $k-1$ boundaries between I_i and I_{i+1}, $1 \le i \le k-1$, and these items share at most one common B_j across the boundary. So the total number of shared bins is at most $k-1$. □

For any packing P, we now define a "deviation" d_i for each I_i by $d_i = h_P(I_i) - \tau$. We denote the sequence $(d_1, \ldots, d_m)$ of deviations by D.

Claim 3 For the canonical packing C, for $1 \le k \le m$,

$$\sum_{i=1}^{k} d_i \le k-1 \tag{3}$$

Proof.

$$\begin{aligned}
\sum_{i=1}^{k} d_i &= \sum_{i \le k} h_C(I_i) - k\tau \\
&\le h_C(I_1 \cup \ldots \cup I_k) + k - 1 - k\tau \qquad \text{by Claim 2} \\
&= k(\frac{1}{k} h_C(I_1 \cup \ldots \cup I_k) - \tau) + k - 1 \\
&\le k(\lceil \frac{1}{k} h_C(I_1 \cup \ldots \cup I_k) \rceil - \tau) + k - 1 \\
&\le k-1 \qquad \text{by def. of } \tau
\end{aligned}$$

□

Claim 4 Suppose for any packing P, $D = (d_1, d_2, \ldots, d_m)$ satisfies (3), and let $D' = (d_1, d_2, \ldots, d_{j-1}, d_{j+1}, \ldots, d_m) = (d'_1, d'_2, \ldots, d'_{m-1})$ be formed from D by deleting some term $d_j \ge 1$. Then D' satisfies (3).

Proof. For $k \le j-1$, we have

$$\sum_{i \le k} d'_i = \sum_{i \le k} d_i \le k-1$$

by the hypothesis on D. For $k \ge j$, we have

$$\begin{aligned}
\sum_{i \le k} d'_i &= \sum_{i \le j-1} d'_i + \sum_{j \le i \le k} d'_i \\
&= \sum_{i \le j-1} d_i + \sum_{j \le i \le k+1} d_i - d_j \\
&= \sum_{i \le k+1} d_i - d_j \\
&\le k - d_j \\
&\le k-1
\end{aligned}$$

by the hypothesis on D. □

4 The algorithm

We now give a packing algorithm that will give a near-optimal solution requiring at most $\tau + 2$ bins for each I_i. In the next subsection, we describe a procedure referred to as "cross-splicing", which will be used by the algorithm. The description of the main procedure of the algorithm follows.

4.1 Cross-splicing

Let us say that the sequence of deviations D is *reduced* if $d_i \notin \{1, 2\}$ for any i. Suppose for some $i < j$,

$$h(I_i) = \tau - a, \quad h(I_j) = \tau + b$$

where $a \geq 0, b \geq 3$.

Let us line up I_j below I_i (see Figure 2) and define the following function:

$$\Delta(x) := h(I_j, x) - h(I_i, x)$$

where $h(I_j, x) :=$ number of different bins that I_j has hit up to x (with $h(I_i, x)$ defined

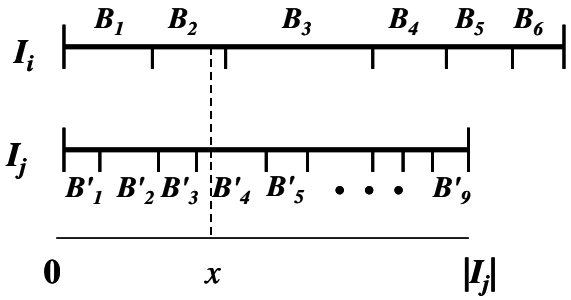

Fig. 2. Lining up I_i and I_j

similarly). By convention $\Delta(0) = 0$ (If we want, we can imagine that bin intervals are of form $(\ldots]$, i.e., semi-open intervals closed on the right.)

Note that $\Delta(x)$ only changes as x crosses a bin boundary, and consequently, it can change (up or down) by at most 1. If I_i and I_j have a *common* boundary value at some point x_0, then $\Delta(x)$ does not change as x goes through x_0. Also note that

$$\Delta(|I_j|) \geq a + b$$

This follows from the fact that since $i < j$, then $|I_i| \geq |I_j|$. Thus, $h(I_i, |I_j|)$ is less than or equal to $\tau - a$.

For an arbitrary value c, $0 < c \leq a+b$, let x_0 be the *first* bin interval right-hand endpoint with $\Delta(x_0) = c$. A cross-splice at x_0 is the following modification: The portion of I_i and I_j

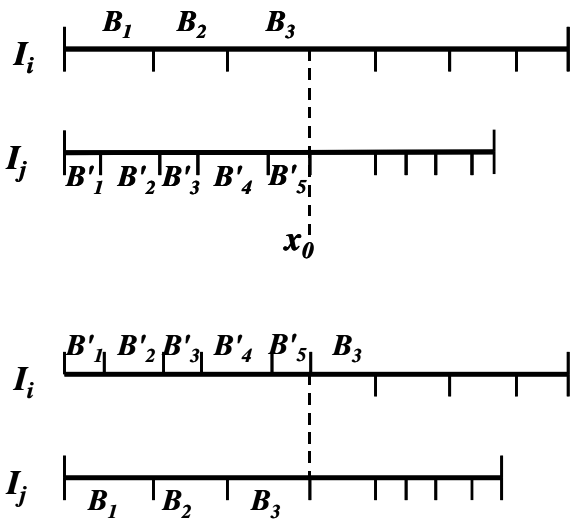

Fig. 3. Cross-splicing at x_0

for $0 \leq x \leq x_0$ are interchanged, as shown in Figure 3. In this example, $c = 2$. The first right-hand endpoint of a bin interval, for which $x_0 = 2$, is that of B'_5. Hence, B'_1 through B'_5 are interchanged with B_1, B_2, and part of B_3. Of course in doing so, we have split bin B_3 between I_i and I_j.

This obviously changes the values of $h(I_i)$ and $h(I_j)$. Normally, x_0 will not be a boundary point for I_i, and in this case, we see that

$$h'(I_i) = h(I_i) + c + 1$$
$$h'(I_j) = h(I_j) - c$$

In the example of Figure 3, I_i initially hits 6 bins, and I_j hits 10 bins. After the cross-splicing, I_i hits 9 bins, and I_j hits 8.

Otherwise, if x_0 is also a boundary point for I_i then

$$h'(I_i) = h(I_i) + c$$
$$h'(I_j) = h(I_j) - c$$

Thus, the D sequence goes from $(d_1, d_2, \ldots, d_i, \ldots, d_j, \ldots, d_m)$
to $(d_1, d_2, \ldots, d_i + c + 1, \ldots, d_j - c, \ldots, d_m)$

4.2 The main procedure

We begin with the canonical (decreasing) packing C explained in Section 3. Let the D sequence for the packing be $(d_1, d_2, \ldots, d_i, \ldots, d_j, \ldots, d_m)$, which satisfies (3) and which we can assume is reduced (i.e., no $d_i = 1$ or 2). Let j be the least index such that $d_j \geq 3$. If there is no such index, then the algorithm completes, and all the items have $h(I_i)$ less than or equal to $\tau + 2$, which is what the algorithm set out to guarantee. By putting $k = 1$ in equation (3), we have $d_1 \leq 0$. Thus, $j \geq 2$. Hence, we have two candidates, I_1 and I_j, that we use for cross-splicing.

We use the same symbols as in the previous subsection, $d_1 = -a, d_j = b, a \geq 0, b \geq 3$. Now, there are two cases:

Case (i) $b > a$. Then we cross-splice I_1 and I_j using $c = a + 1 \leq a + b$. This produces $D' = (2, \ldots, b - a - 1, \ldots, d_m)$, i.e., $d'_1 = 2, d'_j = b - a - 1$. Then we reduce D' to $D'' = (d_2, d_3, \ldots, d'_j, d_{j+1}, \ldots, d_m)$ by removing the entry $d'_1 = 2$.

Case (ii) $b \leq a$. In this case we cross-splice I_1 and I_j using $c = b - 2 \leq a + b$. This produces $D' = (b - a - 1, \ldots, 2, \ldots, d_m)$ which we reduce to $D'' = (b - a - 1, \ldots, d_{j-1}, d_{j+1}, \ldots, d_m)$ by removing the entry $d'_j = 2$.

Claim 5 The resulting reduced sequence D'' satisfies (3).

Proof. We shall first prove this for *Case (i)* described above. For $1 \leq k \leq j - 2$,

$$D'' = (d_2, \ldots, d_{j-1}, b - a - 1, d_{j+1}, \ldots) = (d''_1, d''_2, \ldots)$$

Since by definition, j was the least index such that $d_j > 2$, then $d_2 \leq 0, \ldots, d_{j-1} \leq 0$. We recall that they cannot be equal to 1 or 2 since the sequence is assumed to be reduced. Thus, $\sum_{i \leq k} d''_i \leq 0 \leq k - 1$.

For $k \geq j - 1$ we have

$$\begin{aligned}
\sum_{i \leq k} d''_i &= \sum_{2}^{j-1} d_i + b - a - 1 + \sum_{i=j}^{k} d''_i \\
&= \sum_{i=1}^{j} d_i - 1 + \sum_{i=j+1}^{k+1} d_i \\
&= \sum_{i=1}^{k+1} d_i - 1 \\
&\leq k - 1
\end{aligned}$$

by (3).

Now, if $b - a - 1$ equals 1 or 2, then we remove d''_{j-1} from the sequence D'' to get a new sequence D''', which, by Claim 4, still satisfies (3).

We shall now prove that (3) holds for *Case (ii)*. In this case, we have

$$D'' = (b - a - 1, d_2, \ldots, d_{j-1}, d_{j+1}, \ldots) = (d''_1, d''_2, \ldots)$$

Since $b \leq a$ in this case then as before, $d''_1 \leq 0, \ldots, d_{j-1} \leq 0$, so that for $k \leq j - 1$,

$$\sum_{i \leq k} d''_i \leq 0 \leq k - 1$$

If $k \geq j$, then

$$\begin{aligned}
\sum_{i \leq k} d''_i &= \sum_{i \leq j-1} d''_i + \sum_{i \geq j}^{k} d''_i \\
&= b - a - 1 + \sum_{i=2}^{j-1} d_i + \sum_{i=j+1}^{k+1} d_i \\
&= \sum_{i=1}^{k+1} d_i - 1 \\
&\leq k - 1
\end{aligned}$$

by (3). □

We note that (3) $\Longrightarrow d_1 \leq 0$. So now the algorithm can iterate on the new sequence D'' or D'''. Thus, by cross-splicing, we can reduce the number of $d_i \neq 1$ or 2 by at least one at each step. Strictly speaking, we cross-splice and delete the 2 to get a sequence D'' which (still) satisfies (3). Values of $d_i = 1$ or 2 correspond to I_i which are "happy", i.e., they do not have to be processed any further. Therefore, in at most $m - 1$ steps we have a stable sequence D^* with all entries ≤ 0 (or which is empty). At this point, the algorithm halts.

4.3 Running time

There are two main parts to the algorithm. The first is the sorting of I and B to yield the canonical decreasing packing C, which takes $O(n \log n)$ time. The second involves the iterative cross-splicing phase. The number of iterations is at most $m - 1$. Within each iteration, finding candidate items for cross-splicing takes at most m steps, while computing Δ values at bin endpoints will take $O(n)$ time. Thus, the running time of this part of the algorithm is $O(mn)$. Hence overall, the time complexity of the algorithm is $O(n(m+\log n))$.

	τ	$\tau+1$	$\tau+2$
No cross-splicing reqd.	237	7474	25296
Cross-splicing reqd.	0	0	66993

Table 1. Results obtained from experimental evaluation of our algorithm on 100,000 simulated systems.

4.4 Experiments

We performed an experimental evaluation of this algorithm, by simulating a system with 100 items and 6000 bins. The item sizes followed a uniform random distribution with a mean of 1000 units, and the bin sizes followed a Zipf distribution with $\alpha = 1$ and a maximum size of 50 units.

We ran the simulation 100,000 times, each time with different sets of item and bin sizes. The first row of Table 1 shows the number of simulated systems that did not require any cross-splicing, *i.e.*, the canonical packing provided a solution within $\tau+2$. This is subdivided into the number of systems for which the solution was exactly τ, $\tau+1$, and $\tau+2$. The second row provides the same information for the number of systems that required the iterative cross-splicing phase.

The numbers show that cross-splicing always provides a solution that is exactly $\tau + 2$. This is because each iteration of cross-splicing always creates at least one item that hits $\tau + 2$ bins.

5 Summary

In this paper, we have defined a specific bin-packing problem, that of "locality-maximizing assignment", which is relevant to current distributed applications. We have shown that obtaining the optimal solution is an NP-hard problem, but that an efficient near-optimal algorithm exists. We describe one such algorithm and prove that it provides solutions within an additive constant of 2 of the optimal solution. Finally, we provide empirical data obtained from the experimental evaluation of the algorithm.

References

[Boi] Berkeley open infrastructure for network computing (boinc). http://boinc.ssl.berkeley.edu/.

[DKK+01] F. Dabek, M. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with cfs. In *proceedings of the 18th ACM Symposium on Operating System Principles (SOSP)* , 2001.

[Ent] Entropia website. www.entropia.com.

[FK99] Ian Foster and Carl Kesselman, editors. *The Grid: Blueeprint for a New Computing Infrastructurre.* Morgan Kaufmann, 1999.

[GJ75] M. R. Garey and D. S. Johnson. Complexity results for multiprocessor scheduling under resource constraints. *SIAM Journal of Computing*, 4:397–411, 1975.

[Kar72] R. M. Karp. *Complexity of Computer Computations*, chapter Reducibility among combinatorial problems, pages 85–103. Plenum Press, New York, 1972.

[KBC+00] John Kubiatowicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westly Weimer, Christopher Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*, 2000.

[MMGC02] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: A read/write peer-to-peer file system. In *Proceedings of 5th Symposium on Operating Systems Design and Implementation*, 2002.

[RD01] A. Rowstron and P. Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP'01)*, Oct 2001.

[Set] Search for extra-terrestrial intelligence(seti). http://setiathome.ssl.berkeley.edu/.